

## Algoritmos y Estructuras de Datos II

### Segundo parcial – Sábado 22 de junio de 2019

Orden: 90

A+++

- El parcial es a libro abierto.
  - Cada ejercicio debe entregarse en hojas separadas.
  - Incluir en cada hoja el número de orden asignado, número de hoja, apellido y nombre.
  - Antes de entregar, remover los "pelitos" del borde de las hojas, si hubiere.
  - Cada ejercicio se calificará con Perfecto, Aprobado, Regular, o Insuficiente.
  - El parcial estará aprobado si el ejercicio 1 tiene A o P, y al menos uno de los dos ejercicios restantes tiene A o P.
- Los ejercicios no se recuperan por separado.

I	II	III
P	P	P

(Algoritmo)

Invente una nueva hoja.

### Ej. 1. Diseño

En una elección intervienen *agrupaciones* políticas, numeradas desde 1 hasta  $G$ . Cada agrupación nuclea a varios *candidatos* en una lista electoral. Cada candidato se identifica por un número natural. Durante la jornada de elecciones los votantes registran sus votos en *mesas* electorales. Los votantes se identifican por su DNI, y las mesas se identifican por su nombre, que es un string de largo  $M$ . Al finalizar los comicios, cada mesa registra el resultado del recuento en el Centro de Procesamiento de Datos (CPD), que sumaria la información. Los datos que envía la mesa incluyen el conjunto de votantes que votaron en esa mesa y el número de votos para cada agrupación. Se desea diseñar el CPD para cumplir con los requerimientos detallados abajo. **Importante:** una mesa puede registrar varias veces los correspondientes datos al CPD, con el fin de corregir posibles errores u omisiones. Cada vez que una mesa registra sus datos en el CPD, toda la información que esa mesa hubiera enviado anteriormente se descarta por completo, y se sobrescribe con la nueva información.

TAD CANDIDATO es NAT	TAD AGRUPACIÓN es NAT	TAD DNI es NAT	TAD MESA es STRING
TAD CPD			
observadores básicos			
agrupaciones	: cpd	→ conj(agrupación)	
candidatos	: cpd $c \times$ agrupación $a$	→ secu(candidato)	$\{a \in \text{agrupaciones}(c)\}$
votantes	: cpd $\times$ mesa	→ conj(dni)	
votos	: cpd $\times$ mesa	→ multiconj(agrupación)	
generadores			
iniciar	: dicc(agrupación $\times$ secu(candidato))	→ cpd	
registrar	: cpd $c \times$ mesa $\times$ conj(dni) $ds \times$ multiconj(agrupación) $as$	→ cpd	$\{\#(ds) = \#(as) \wedge (\forall a : \text{agrupación})(a \in as \Rightarrow a \in \text{agrupaciones}(c))\}$
axiomas			
...			
Fin TAD			

Se debe realizar un diseño que provea las siguientes operaciones con las complejidades en peor caso indicadas:

- REGISTRAR(*inout*  $c$  : CPD, *in*  $m$  : MESA, *in*  $ds$  : CONJ(DNI), *in*  $as$  : MULTICONJ(AGRUPACIÓN)) — Registrar la información de una mesa —  $O(M + G + V_A \cdot \log V)$ , donde  $V$  es la cantidad total de votantes que hay en el sistema y  $V_A$  es la cantidad total de votantes que son *afectados* por esta operación, incluyendo a todos los votantes del conjunto  $ds$  y a todos aquellos que tuvieran votos registrados anteriormente en la mesa  $m$ . Recordar que esta operación debe *descartar* toda la información que hubiera previamente registrada en la mesa  $m$ .
- VOTOSPARA(*in*  $c$  : CPD, *in*  $x$  : CANDIDATO) → *res* : NAT — Determinar el número de votos que recibió el candidato  $x$ , sumando los votos que recibieron todas las agrupaciones en las que aparece. (Notar que un candidato puede figurar en las listas de varias agrupaciones distintas). —  $O(G + \log K)$ .
- REPETIDOS(*in*  $c$  : CPD) → *res* : CONJ(DNI) — Obtener el conjunto de DNIs de los votantes que tengan votos registrados en dos o más mesas diferentes. Observar que este conjunto puede "achicarse" si una mesa registra una versión corregida de sus datos. —  $O(1)$ .

donde  $M$  es la longitud de los nombres de las mesas,  $G$  es el número de agrupaciones (recordar que las agrupaciones se numeran desde 1 hasta  $G$ ) y  $K$  es la cantidad total de candidatos.

1. Dar una estructura de representación del módulo CPD explicando detalladamente qué información se guarda en cada parte, las relaciones entre las partes, y las estructuras de datos subyacentes.
2. Justificar de qué manera es posible implementar los algoritmos para cumplir con las complejidades pedidas. Escribir el algoritmo para la operación REGISTRAR.

## Ej. 2. Ordenamiento

Sea  $A[0..n-1]$  un arreglo de  $n$  números naturales distintos entre sí. Dado un número  $k \in \{1, \dots, n\}$  decimos que el arreglo  $A$  está  $k$ -ordenado si para todo índice  $i < n$  se tiene que el  $i$ -ésimo elemento más chico aparece en el arreglo antes de la posición  $i + k$ . Más precisamente, sea  $B[0..n-1]$  el arreglo  $A$  ordenado de menor a mayor. Entonces:

$$A \text{ está } k\text{-ordenado} \iff (\forall i : \text{nat})(i < n \Rightarrow (\exists j : \text{nat})(j < \min(i + k, n) \wedge B[i] = A[j]))$$

Dicho de otro modo, los primeros  $m$  elementos de  $B$  se pueden encontrar en el prefijo de  $A$  que tiene tamaño  $m + k$ .

Por ejemplo, el arreglo  $A = [5, 2, 4, 7, 6]$  está 3-ordenado porque el arreglo ordenado es  $B = [2, 4, 5, 6, 7]$ , de tal forma que el elemento  $B[i]$  aparece en la posición  $j$  de  $A$  con  $j < i + 3$ :

$i$	$j$	$< i + 3$
0	1	$< 3$
1	2	$< 4$
2	0	$< 5$
3	4	$< 6$
4	3	$< 7$

Notar que si el arreglo  $A$  está 1-ordenado entonces está ordenado de menor a mayor.

Suponiendo que  $A$  está  $k$ -ordenado, se pide proponer un algoritmo para ordenarlo con costo  $O(n \log k)$  en peor caso. Justificar adecuadamente su complejidad.

**Nota:** quizás puede ayudar pensar primero en el caso en el que  $A$  está 2-ordenado. En tal caso el algoritmo debería ser  $O(n)$ .

## Ej. 3. Dividir y Conquistar

En este ejercicio llamamos **cadena** a una secuencia que empieza en un número  $x$  y tiene  $k \geq 1$  elementos, cada uno de los cuales es el doble del anterior, es decir, una cadena es una secuencia de la forma  $[x, 2x, 4x, \dots, 2^{k-1}x]$ . Dado un arreglo  $A[0..n-1]$  de  $n$  números naturales ordenados estrictamente de menor a mayor (sin repetidos), se quiere implementar una operación:

$$\text{LONGITUDCADENAMÁS LARGA}(\text{in } A : \text{ARREGLO}(\text{NAT})) \rightarrow \text{NAT}$$

para determinar la longitud de la subsecuencia<sup>1</sup> del arreglo  $A$  que determina la cadena más larga.

Por ejemplo, en el arreglo ordenado  $A = [2, 3, 4, 6, 8, 12, 16, 18, 32, 36]$  la longitud de la cadena más larga es 5 (y sus elementos son  $[2, 4, 8, 16, 32]$ ). Notar que  $[4, 8, 16]$  y  $[3, 6, 12]$  también son cadenas del arreglo  $A$ , pero no tienen la longitud más larga posible. Notar también que podría haber varias cadenas que "empaten" de tal modo que todas tengan la longitud más larga posible.

Proponer un algoritmo que implemente LONGITUDCADENAMÁS LARGA. La complejidad temporal del algoritmo debe ser estrictamente menor que  $O(n^2)$  en peor caso. Determinar y justificar la complejidad del algoritmo propuesto.

**Sugerencia:** representar una cadena  $[x, 2x, 4x, \dots, 2^{k-1}x]$  como una tripla  $(b, p, q)$  cuya primera componente  $b$  es un número impar tal que el extremo izquierdo de la cadena es  $x = 2^p \cdot b$  y el extremo derecho de la cadena es  $2^{k-1}x = 2^q \cdot b$ , es decir,  $q - p = k - 1$ . Por ejemplo, la cadena  $[12, 24, 48]$  se puede representar como la tripla  $(3, 2, 4)$ . Suponer que se cuenta con una función  $\text{DESCOMPONER} : \text{Nat} \rightarrow (\text{Nat}, \text{Nat})$ , con costo  $O(1)$  en peor caso, tal que  $\text{DESCOMPONER}(x) = (b, p)$  donde  $b$  es un número impar y  $x = 2^p \cdot b$ . Por ejemplo,  $\text{DESCOMPONER}(12) = (3, 2)$ .

<sup>1</sup> Recordar que una subsecuencia es una secuencia de elementos no necesariamente consecutivos del arreglo.

Martín Ariel LU: 208118 Excelente parcial!

Schuster DNI: 41471933

(AHH) Hoja 1 Orden: 90  
(Alexis)

### Ejercicio 1:

CPD se representa con  $estr$ , donde  $estr$  es

Tupla  $\langle mesas: \text{diccString}(mesa, \langle vot: \text{conjLineal}(DNI), vota: \text{arreglo}(nat)[G] \rangle)$

$agrupxCand: \text{diccLog}(\text{candidato}, \text{conj}(\text{agrupación})),$

$VOTOSxAgrup: \text{arreglo}(nat)[G],$

$votantesTOT: \text{diccLog}(DNI, \langle \#AP: nat, rep: \text{itConjLineal} \rangle),$

$repetidos: \text{conjLineal}(DNI) \rangle$

### Explicación:

Mesas son las mesas en las que se votó. Para cada una de ellas usamos un  $\text{diccString}$  (explicado más adelante) que, dada una mesa, devuelve el conjunto de votantes de esa mesa junto con un arreglo que representa los votos. Para cada partido, siendo un partido su posición en el arreglo.

AgrupxCand es un  $\text{diccLog}$  (explicado más adelante) que permite saber, dado un candidato, el conjunto de agrupaciones a las que pertenece.

VOTOSxAgrup Es un arreglo similar al descrito en Mesas pero que contiene la cantidad total de votos en cada agrupación.

repetidos es un conjunto que posee aquellos DNIs que tengan votos en 2 o más mesas diferentes.

VotantesTOT: Es un  $\text{diccLog}$  que, dado un DNI, devuelve una tupla con la cantidad de apariciones en mesas distintas y un iterador a  $repetidos$  que apunta a ese mismo DNI en el otro conj si  $\#AP > 1$

Explicación Dice log, Dice String & Multicon:

Ambas estructuras son diccionarios montados sobre alguna estructura que nos permita ~~hacer~~ ~~en~~ ~~las~~ hacer las operaciones del diccionario en las complejidades pedidas. En el caso del dicLog, podemos usar un AVL para poder insertar, acceder y borrar en  $O(\log(n))$  siendo  $n$  la #claves. En el caso del dicString, podemos usar un trie para hacer esas mismas operaciones en  $O(l)$  siendo  $l$  un string y  $l$  su longitud.

Para el multicon, entiendo que se puede implementar sobre una lista  
a entrada (con notación con valor repetido) y que se puede tomar  
~~se puede tomar~~ UN ITERADOR al módulo (librecorre, un iterador  
a la lista)

Algoritmos:

## Algoritmos:

VOTOS Para: Buscamos al candidato en  $e.agrupxCand$  en  $O(\log(n))$ . Luego, iteramos en  $e.votosxAgrup$  en  $O(1)$  (por estructura de arreglo) y ~~se~~ por cada agrupación asociada al candidato ~~se~~ sumamos el contenido de dichas posiciones. Obs: Pedimos que el obtener de  $e.agrupxCand$  sea por ref para hacerlo en  $O(1)$ . ~~se~~ Por caso  $O(1)$  si un candidato está en todas las

Complejidad:  $O(\log(n) + G)$

✶ Peor caso (UG)  
Si un candidato  
está en todas las  
agrupaciones.

Repetidos:

Devolveremos por ref e repetidos. /

Complexity:  $O(1)$ .



Martin Ariel

LU: 208/18

Schuster

DNI: 41471933

orden 90

Hoja 2

IR Registrar (input C: CPO, in m: mesa, in ds: conj(DNI), in as: multiconj(Agrupación))

res servicios  
usados

INFO M  $\leftarrow$  obtener(m, C. mesas) // por ref O(M)

for (i = 0 ... G) do // O(G)

C. VOTOS x Agrup [i] = C. votos x Agrup [i] - INFO M. VOTA [i] // O(1)

INFO M. VOTA [i] := 0 // O(1)

endfor

~~for i = 0 ... G~~

IT  $\leftarrow$  crear IT (as) // O(1)

while (hay Siguiente (IT)) do // O(V<sub>A</sub>)

INFO M. VOTA [siguiente (IT)] ++ // O(1)

C. VOTOS x Agrup [siguiente (IT)] ++ // O(1)

endwhile

ITVOT  $\leftarrow$  crear IT (INFO M. VOT) // O(1)

while (hay Siguiente (ITVOT)) do // O(V<sub>A</sub> log(V))

VTOT  $\leftarrow$  obtener (siguiente (ITVOT), C. VOTES TOT) // O(log(V)) por ref

VTOT. #AP --

endwhile

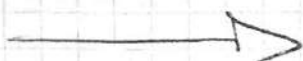
ITNuevosVOT  $\leftarrow$  crear IT (ds) // O(V<sub>A</sub> log(V))

while (hay Siguiente (ITNuevosVOT)) do

NUVOT  $\leftarrow$  obtener (siguiente (ITNuevosVOT), C. VOTES TOT) // O(log(V))

NUVOT. #AP ++

endwhile



ITVOT ← crear IT (INFO.M. VOT)

```
while (hay siguiente(ITVOT)) do // O(Av)
  i ← obtener(siguiente(ITVOT), C.VOTANTESOT) // O(log(V)) por ref
  IF siguiente(ITVOT).#AP > 1
    IF siguiente(ITVOT).rep ≠ Null then
      iR ← Ag Añs (C.repeticiones, siguiente ITVOT) // O(1)
      i.rep ← iR // O(1)
    endif
  else IF i.rep ≠ Null then
    eliminar siguiente(i.rep) // O(1)
    i.rep ← Null // O(1)
  endif
endif
endwhile
```

$O(Av \log(V))$

ITVOT ← crear IT (ds)

•) Hago exactamente lo mismo que ~~en~~ arriba pero con los nuevos votantes. //  $O(V_A \log(V))$

Recorro los votantes nuevos y viejos y actualizo los repetidos

\*1

↳ con ITVOT

INFO.M. VOT ← ds //  $O(V_A)$ .

actualizo votantes de la mesa.

Complejidad :  $O(Av \log(V) + M + G)$

### Servicios Usados

• Como servicio usado le pido al obtener que si el elem no está definido, lo defina con el it en Null y #AP en 0 y me devuelva una referencia hacia esa definición. En el caso del diceString, le pido que genere un Arreglo de tamaño G con todas las POS en 0 y el conj de DNI vacío.

Obs: La complejidad de G no molesta pues se la considera.

↳ si el elem no estaba definido, definirlo cuesta  $O(M+G)$ .

Martin Ariel LU: 208/18  
Schuster DNI: 41471933

Orden 90  
Hoja 3

## Ejercicio 2:

Idea: Hago un minheap de  $n$  elementos y voy encolando y desencolando los elementos por cada posición avanzada.

$n$ -Ordenado (in  $A$ : Arreglo ( $nat$ ), in  $n$ :  $nat$ )  $\rightarrow$  Arreglo ( $nat$ )  
 $res \leftarrow Arreglo(nat)[n] // O(n)$

$heap \leftarrow Arreglo(nat)[n] // O(n)$

Array 2<sup>min</sup> Heap Floyd ( $A[0, n]$ ,  $heap$ )  $// O(n)$

~~for  $i = 0$  to  $n$  do~~

~~encolar( $A[i]$ ,  $heap$ )~~

~~if  $i + n \leq n$  then~~

~~desencolar( $A[i]$ ,  $heap$ )~~

~~ponerlos en el arreglo~~

for ( $i = 0, \dots, n$ ) do  ~~$O(\log n)$~~   $O(\log n)$

if  $i + n \leq n$  then

encolar( $A[i+n]$ ,  $heap$ )  $// O(\log n)$

endif

~~desen~~  $res[i] \leftarrow desencolar(heap) // O(\log n)$

endfor

Complejidad:  $O(n \log n)$

Justificación: El algoritmo heapify de Floyd cuesta  $O(\#elem)$ , en este caso  $O(n) \in O(n \log n)$ .

Crear los arreglos es  $O(n+n) \in O(n \log n)$ .

Encolar y desencolar en un heap es  $O(\log n)$ ; Luego, hacerlo  $n$  veces cuesta  $O(n \log n)$ .

Martín Ariel LU: 208/18  
Schuster DNI: 41471933

Order 90  
Hoja 4

### Ejercicio 3:

Longitud Cadena Más Larga (in A: Arreglo (nat))  $\rightarrow$  nat

$l \leftarrow \text{LongAux}(A)$

$\text{max} \leftarrow 0 \quad // O(1)$

$it \leftarrow \text{creaIT}(l) \quad // O(1)$

while (haySig(it)) do  $// O(n)$

~~if sig long sig~~

if  $\frac{\pi_3(\text{sig}(l)) - \pi_2(\text{sig}(l)) + 1}{2} > \text{max}$   
then  $\text{max} \leftarrow$

end if

endwhile

ret max

end function.

LongAux (in A: Arreglo (nat))  $\rightarrow$  lista (tupla (nat, nat, nat))

if  $\text{long}(A) == 1$  then

$d \leftarrow \text{descomponer}(A[0]) \quad // O(1)$

ret  $\langle \pi_1(d), \pi_2(d), \pi_2(d) \rangle \quad // O(1)$

Caso base  
 $O(1)$

endif

$izq \leftarrow \text{LongAux}(A[0, \frac{\text{long}(A)}{2}])$

$der \leftarrow \text{LongAux}(A[\frac{\text{long}(A)}{2}, \text{long}(A)])$

$l \leftarrow \text{vacía}() \quad // O(1)$

$IT_{izq} \leftarrow \text{creaIT}(izq) \quad // O(1)$

$IT_{der} \leftarrow \text{creaIT}(der) \quad // O(1)$



while (hay Sig (l, l2q)  $\wedge$  hay Sig (l, Der)) do  $O(n)$

Mejoro y mantengo el orden por 1er criterio  
 $i \leftarrow \text{sig}(l, l2q) // O(1)$   
 $j \leftarrow \text{sig}(l, Der) // O(1)$  He fijo que no se permita una secuencia como [2, 8, 16]

IF  $\pi_1(i) = \pi_1(j) \wedge \pi_2(j) = \pi_3(j) + 1$  then  $O(1)$

AgAtras (l,  $\langle \pi_1(l2q), \pi_2(i), \pi_3(j) \rangle$ )  $O(1)$

Avanzar (l, l2q)  
 Avanzar (l, Der)  $O(1)$

else

IF  $\pi_1(l2q) < \pi_1(Der)$  then

AgAtras (l, i)

Avanzar (l, l2q)

else

AgAtras (l, j)

Avanzar (l, Der)

$O(1)$

$O(1)$

endif

endwhile

concat (l, l2q)  $O(1)$  ] Agrego lo que haya faltado  
 concat (l, Der)  $O(1)$

set l  $O(1)$

endfunction

## Complejidad:

Como vemos en LongAprox que dividimos en 2 subproblemas con la mitad del tamaño del Array Para cada una, podemos decir que  $a=c=2$ . Luego, vemos que el merge cuesta  $O(n)$

Puesto que queremos mergear y mantener el orden respecto de la primer componente. Luego, vemos que  $T(n) = 2T(n/2) + f(n)$  con  $f(n) \in O(n)$ .

Luego, estamos en el 2do caso del teo maestro  $\rightarrow$  porque  $\log_2 2 = 1$  ya que  $f(n) \in O(n)$ . Finalmente, nuestra complejidad es

$$O(n \log(n)) < O(n^2)$$